Севастопольский государственный университет
Институт информационных технологий и управления в технических системах

Сети глубокого обучения и компьютерное зрение

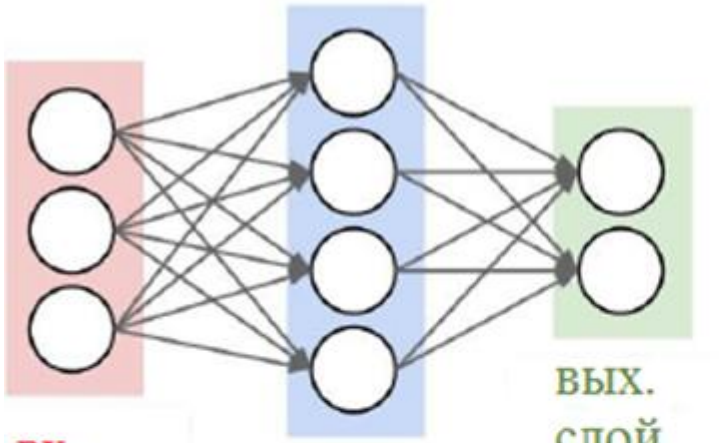
Бондарев Владимир Николаевич

Севастопольский государственный университет
Институт информационных технологий и управления в технических системах

ЛР5: Полносвязанные нейросети

Бондарев Владимир Николаевич

Двухслойная нейросеть



"2-х слойная сеть"
или "сеть с 1-м
скрытым слоем"

“Fully-connected” слои
Полносвязные слои

Модульная реализация многослойных ИНС

Мы хотим создавать сети с использованием модульного подхода, чтобы мы могли изолировать отдельные типы слоев, а затем объединить их в модели с разными архитектурами.
Для каждого слоя мы реализуем функции прямого **forward** и обратного **backward** распространения.

```
def layer_forward(x, w):
 """ Принимает на вход x и веса w """
 # Некоторые вычисления ...
z = # ... внутренние значения
 # Дополнительные вычисления ...
out =  # выход
cache = (x, w, z, out)
 # Значения, которые нужны для вычисления градиентов
 return out, cache
```

Модульная реализация многослойных ИНС

Функция обратного распространения будет получать производные восходящего потока и объект cache и будет возвращать градиенты по отношению к входам и весам, например:

```
def layer_backward(dout, cache):
""" Принимает dout (производная функции потерь по выходу) и кеш, и вычисляет производные по отношению ко входам. "“”

x, w, z, out = cache # Извлечение значений из кэша
# Используем значения из кэша для вычисления производных
 dx = # Производная потерь по x
dw = # Производная потерь по w
 return dx, dw
```

В дополнение к реализации полносвязанных сетей произвольной глубины мы также изучим различные правила обновления (для оптимизации) и также введем **Dropout** в качестве регуляризатора, а **нормализацию на блоке/ слое** в качестве инструмента для более эффективной оптимизации глубоких сетей.

Реализация affine_forward

```
def affine_forward(x, w, b):

"""Входы:
    - x: массив numpy, содержащий входные данные, формы (N, d_1, ..., d_k)
    - w: Множество весовых коэффициентов, формы (D, M)
    - b: Массив смещений, формы (M,)

    Возвращает кортеж из:
    - out: выход, формы (N, M)
    - cache: (x, w, b) """

N = x.shape[0]
 out = x.reshape(N, -1).dot(w) + b
cache = (x, w, b)
 return out, cache
```

Реализация affine_backward

```
def affine_backward(dout, cache):
    """
     - dout: восходящая производная, форма (N, M)
     - cache: кортеж:
       - x: входные данные формы (N, d_1, ... d_k)
       - w: веса формы (D, M)
       - b: смещения формы (M,)
    Возвращает кортеж:
     - dx: градиент по x, форма (N, d1, ..., d_k)
     - dw: градиент по w, форма (D, M)
     - db: градиент по отношению к b, форма (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    N = x.shape[0]
    x_flat = x.reshape(N, -1)
    dx = dout.dot(w.T).reshape(x.shape)
    dw = x_flat.T.dot(dout)
    db = dout.sum(axis=0)
```

Реализация relu_forward

```
def relu_forward(x):
    """
    Выполняет прямое распространение для слоя блоков ReLU.

     Входные данные:
     - x: входы любой формы

     Возвращает кортеж:
     - out: выход, такой же формы, как x
     - cache: x
    """
    out = None
    out = np.maximum(x,0)
    cache = x
    return out, cache
```

Реализация relu_backward

```
def relu_backward(dout, cache):
    """
    Выполняет обратный проход для слоя из блоков ReLU.

    Входные данные:
    - dout: восходящие производные
    - cache: вход x, такой же формы, как dout

    Возвращает:
    - dx: градиент по x """

    dx, x = None, cache
    dx = dout * (x > 0)
    return dx
```

Реализация класса class TwoLayerNet(object)

```
def __init__(self, input_dim=3 * 32 * 32, hidden_dim=100, num_classes=10,
                 weight_scale=1e-3, reg=0.0):
    self.params = {}
    self.reg = reg

    self.params["W1"] = np.random.normal(0.0, weight_scale, (input_dim, hidden_dim))
    self.params["W2"] = np.random.normal(0.0, weight_scale, (hidden_dim, num_classes))
    self.params["b1"] = np.zeros(hidden_dim, dtype=float)
    self.params["b2"] = np.zeros(num_classes, dtype=float)
```

Реализация класса class TwoLayerNet(object)

```
def loss(self, X, y=None):
    """
    Реализуем метод, используя модульную структуру :
        слой_fc1+relu -> слой_fc2 -> softmax
    """
    out1, cache1 = affine_relu_forward(X, self.params["W1"], self.params["b1"])
    out2, cache2 = affine_forward(out1, self.params["W2"], self.params["b2"])
    scores = out2

    loss, grads = 0, {}
    #вычисляем средние потери для миниблока и матрицу градиентов модуля softmax
    loss, dscores = softmax_loss(scores, y)  # используем готовую функцию
    dout1, grads["W2"], grads["b2"] = affine_backward(dscores, cache2)
    dX, grads["W1"], grads["b1"] = affine_relu_backward(dout1, cache1)
    for w in ["W2", "W1"]:
        if self.reg > 0:
            loss += 0.5 * self.reg * (self.params[w] ** 2).sum()
        grads[w] += self.reg * self.params[w]
    return loss, grads
```

Использование класса Solver

В предыдущем задании операторы обучения моделей были включены непосредственно в саму модель. Следуя модульному подходу, в этом задания мы выделим операторы обучения моделей в отдельный класс.

```
model = TwoLayerNet()

solver = Solver(model, data,
                update_rule='sgd',
                optim_config={ 'learning_rate': 1e-3 },
                lr_decay=0.95, num_epochs=6, batch_size=100,
                print_every=100)
solver.train()
```

```
(Iteration 2601 / 2940) loss: 1.276361
(Iteration 2701 / 2940) loss: 1.111768
(Iteration 2801 / 2940) loss: 1.271688
(Iteration 2901 / 2940) loss: 1.272039
(Epoch 6 / 6) train acc: 0.546000; val_acc: 0.509000
```

Реализация класса FullyConnectedNet

Теперь реализуем полносвязанную сеть с произвольным количеством скрытых слоев.
Просмотрите класс FullyConnectedNet в файле cs231n/classifiers/fc_net.py.
Мы предоставляем его полную реализацию. Просто изучите его и используйте.

1. **Проверка градиента**

```
N, D, H1, H2, C = 2, 15, 20, 30, 10
model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                          reg=reg, weight_scale=5e-2, dtype=np.float64)
```

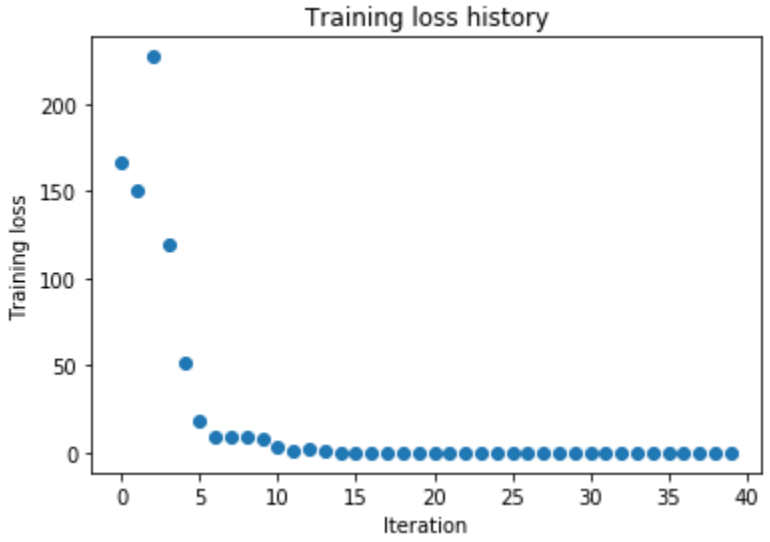
```
Running check with reg = 0
Initial loss: 2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
```

Реализация класса FullyConnectedNet

2. **Проверка переобучением на 50 примерах для 3-х и 5-ти слойной сети**
learning_rate = 2e-3
weight_scale = 1e-1

model = FullyConnectedNet([100, 100, 100, 100], weight_scale=weight_scale, dtype=np.float64)

(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.125000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.039138
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.121000



Правила обновления

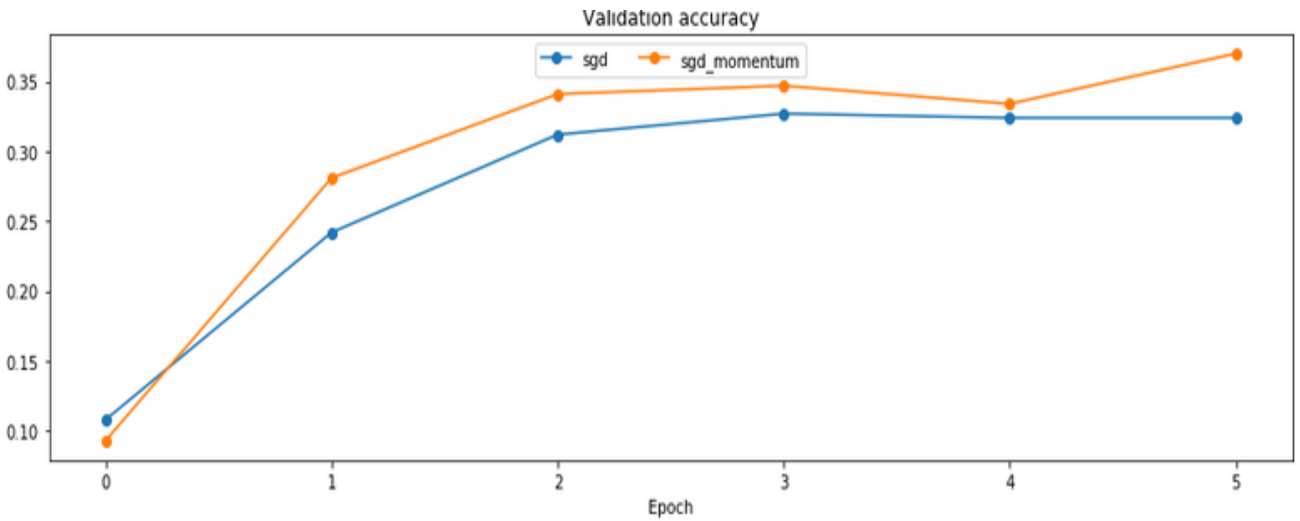
config: словарь, содержащий значения гиперпараметров, такие как скорость обучения, момент и т. д. Если правило обновления требует кеширования значений между итерациями, то **config** будет хранить эти кешированные значения.

```
def sgd_momentum(w, dw, config=None):
    if config is None: config = {}
    config.setdefault('learning_rate', 1e-2)
    config.setdefault('momentum', 0.9)
    v = config.get('velocity', np.zeros_like(w))

    next_w = None

    v=config['momentum'] * v - config['learning_rate'] * dw
     w += v
    next_w = w
    config['velocity'] = v

     return next_w, config
```



Правила обновления

config: словарь, содержащий значения гиперпараметров, такие как скорость обучения, момент и т. д. Если правило обновления требует кеширования значений между итерациями, то config будет хранить эти кешированные значения.

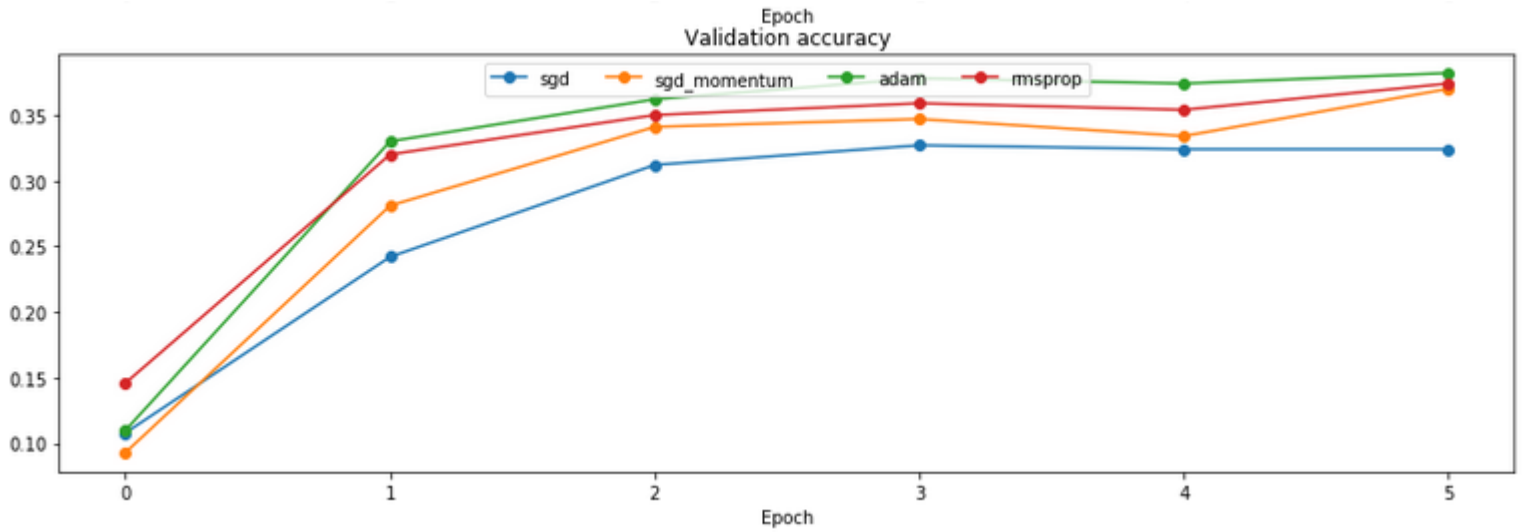
```
def rmsprop(w, dw, config=None):
    ….
    config['cache'] = config['decay_rate'] * config['cache'] + (1 - config['decay_rate']) * (dw**2)
     w += (-config['learning_rate'] * dw )/ (np.sqrt(config['cache']) + config['epsilon'])
     next_w = w
     return next_w, config
```

```
def adam(w, dw, config=None):
    …
    config['t']+=1
    config['m'] = config['beta1']*config['m'] + (1-config['beta1'])*dw
    mt =config['m'] / (1-config['beta1']**config['t'])
    config['v'] = config['beta2']*config['v'] + (1-config['beta2'])*(dw**2)
    vt = config['v'] / (1-config['beta2']**config['t'])
    w += (-config['learning_rate'] * mt) / (np.sqrt(vt) + config['epsilon'])
    next_w = w
    return next_w, config
```

Результаты применения RMSProp и Adam

```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={ 'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solver.train()
```

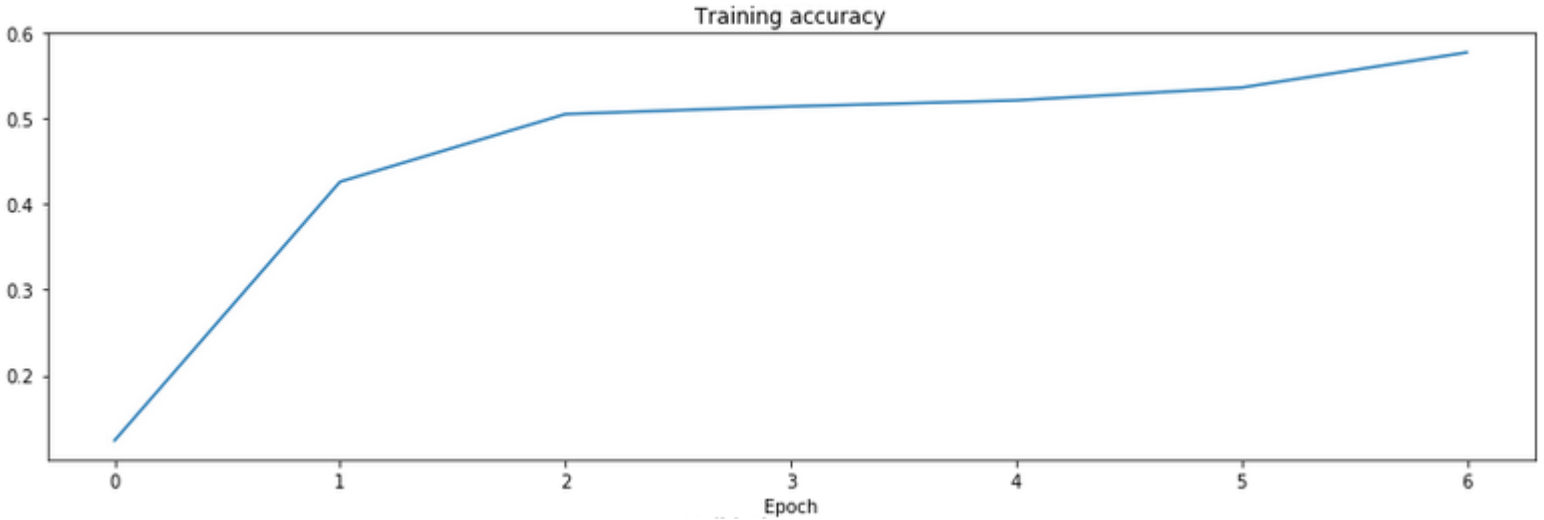


Построение хорошей модели

```
best_model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

solver = Solver(best_model, data,
                num_epochs=6, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3
                },
                print_every=100)

solver.train()
```



Построение хорошей модели

Проверьте модель с блочной нормализацией и dropout. Для этого сначала выполните блокноть BatchNormalization и Dropout.

```
model = FullyConnectedNet(
      [512, 256, 256],
       weight_scale=1e-2,
      normalization='batchnorm',
      dropout=0.5
 )
 solver = Solver(
       model,
      data,
      num_epochs=10,
      print_every=100,
      batch_size=256,
      update_rule="adam",
      optim_config={'learning_rate': 1e-3},
      verbose=True
)
solver.train() best_model = model
```